Definition

- The term “regular expression” was introduced in the lecture on automata
  
  A **regular expression** is a pattern that summarizes a set of strings

- There is a one-to-one correspondence between automata and regular expressions
  
  - a regular expression can be viewed as a specification for how to build a finite automaton (states, transitions, etc)
  
  - given an automaton one can write a regular expression to summarize the strings it recognizes

![Diagram of automaton and regular expression]

Example

- In a regular expression:
  
  - letters stand for themselves
  
  - * means “zero or more of the preceding item”
  
  - | means “or”
  
  - parentheses can be used to make groups

- Example:
  
  - “a followed by b followed by zero or more b’s followed by a followed by any number of ( a or b )”

  ![Diagram of automaton with regular expression example]

  Note: NTO uses + instead of |
Example: Exact Match

- From Lab 2: read a string from the internet (source: NY Times headlines)
  - how long is the string?
  - how many headlines (occurrences of "<B>") does it have?
  - does it contain "Dodgers"?

```ruby
>> s = IO.read("sportsnews.html")
>> s.length
=> 3977
>> s.scan("<B>")
=> ["<B>", "<B>", "<B>", "<B>", "<B>", "<B>"
>> s.index("Dodgers")
=> 3497
```

Example (cont’d)

- These calls to scan and index are examples of exact match searches
- Today we’ll see how to use a regular expression to search for more general patterns and extract more information from the text
- Example: the headline text is between <B> and </B>, e.g.
  `<B>Dodgers’ Reserve Outfielder Drives in 7</B>`

From a loop that uses a regular expression to scan the headlines:

```ruby
>> s.scan(/<B>(.*?)</B>/).each {|p| puts p}
Sports
Low Payroll and High Hopes for Brewers
Tough Park for Hitters? Mets Don’t Think So
Seven-R.B.I. Game May Be
Lilly and Blue Jays Stym
Dodgers’ Reserve Outfield
```

Regular Expression Objects in Ruby

- Regular expressions in Ruby are a special type of string
- Normal string objects are surrounded by double quotes:
  ```ruby
  >> s = "abc"
  => "abc"
  >> s.class
  => String
  ```
- Regular expression objects begin and end with a slash character:
  ```ruby
  >> r = /abc/
  => /abc/
  >> r.class
  => Regexp
  ```

Patterns

- What makes regular expressions so useful is that one expression can match several different strings in the text
- In this case we say the regular expression is a pattern
- We’ve already seen one example: “.” means “any character”
- The same symbol used in “real” regular expressions from automata theory
- Examples:
  - /a.a/ a pattern that matches any 3-letter string starting and ending with “a”
  - /.../ a pattern for any 3-letter string
Regular Expressions in Calls to \texttt{scan}

- Ruby allows you to use a regular expression in a call to \texttt{scan}.
- Examples:
  
  ```ruby
  >> s.scan("<B>")
  => ["<B>", "<B>", "<B>", "<B>", "<B>"]
  >> s.scan(/<B>/)
  => ["<B>", "<B>", "<B>", "<B>", "<B>"]
  >> s.scan(/<..>/)

Sets

- In many cases we'd like to restrict the set of characters that can be in the matching string.
- Example: suppose we want to look for “hep”, “hip”, or “hop” -- strings that start with “h” and end with “p”
  
  ```ruby
  /h.p/ # matches too many strings -- “hap”, “hbp”, “hcp”, etc
  ```
- To restrict the match put the list of allowable letters between square brackets:
  
  ```ruby
  /h[eio]p/ # matches only “hep”, “hip”, or “hop”
  ```
- Mnemonic: think of the labels on a state transition in an FSA

Sets (cont’d)

- Use a dash as a shorthand notation for a range of characters
  
  ```ruby
  [0-9] is the same as \[0123456789\]
  [a-z] is the same as \[abcdefghijklmnopqrstuvwxyz\]
  ```
- Ruby also has special named sets
  
  ```ruby
  \d digit (the same as \[0-9\])
  \w word character (the same as \[a-zA-Z0-9\_\])
  \s space -- matches space, tab, newline
  ```
- Example: search for three-letter words:
  
  ```ruby
  >> s.scan(/\w\w\w/)
  => [" and ", ", for ", ", New ", ...]
  ```
  
  note the space is included with the output -- it’s part of the pattern
  
  doesn’t catch words next to punctuation, e.g. “the old man.”
- The previous slide showed the pattern for a 3-letter word as
  
  ```ruby
  /\s[\w\w]s/ # a word
  ```
  
  Using this format to make a pattern for a 10-letter word is pretty awkward
- Ruby allows you to put a \textit{pattern length} in braces
  
  ```ruby
  /a\{5\}/ # 5 a’s in a row
  /\d\{4\}/ # a four-digit number
  /\w{10}\s/ # a ten-letter word with spaces on both sides
  ```

Pattern Size

```ruby
>> s.scan(/\s\w{10}\s/)
=> [" Outfielder "]
```
Pattern Size (cont’d)

- You can make a range of sizes by separating lower and upper bounds by a comma.
- Words between 5 and 7 letters:
  ```ruby
  s.scan(/\s\w{5,7}\s/)
  #=> 
  ["Payroll ", "Hopes ", "Think ", "Parting ",
  "Stymie ", "Reserve ", "Drives "]
  ```
- Words with 4 or more letters:
  ```ruby
  s.scan(/\s\w{4,}\s/)
  #=> 
  ["Payroll ", "High ", "York ", "2006 ", "Park ", ...
  ```

The * and + symbols from formal regular expressions are also allowed in Ruby:
- * means “0 or more”
- + means “1 or more”
- A question mark means an item is optional

Examples:
- Find the words “COLOR” or “BGCOLOR”:
  ```ruby
  s.scan(/B?G?COLOR/)
  #=> 
  ["BGCOLOR", "BGCOLOR", "COLOR", "BGCOLOR", "COLOR" ...]
  ```
- Find a # followed by one or more digits:
  ```ruby
  s.scan(/#\d+/)
  #=> 
  ["#000000", "#666666", "#666666", "#666666" ...
  ```

Example: HTML Colors

- The #NNNNNN notation is how colors are specified in HTML documents (two digits each for red, green, and blue values)
  ```ruby
  s.scan(/\d{6}/)
  #=> 
  ["#000000", "#666666", "#666666", "#666666", "#000000" ...]
  ```
- But the digits are actually hexadecimal:
  ```ruby
  e.g. #008080 is a valid color
  0% red, 50% green, 66% blue
  ```
- A better pattern:
  ```ruby
  s.scan(/\w{6}/)
  #=> 
  ["#ffffff", "#000000", "#666666", ...
  ```
- This new pattern isn’t strictly accurate:
  ```ruby
  # It allows #AAQQRR, etc
  # What is a more precise pattern specification?
  ```

Anchors

- There are three other special constructs worth knowing:
  ```ruby
  ^ means the pattern must occur at the beginning of a line
  $ means the pattern must be at the end of a line
  \b stands for “word boundary”
  ```
  ```ruby
  s.scan(/\b\w{4}\b/)
  #=> 
  ["<html", "<head", "<title", "<meta", ...
  ```

How would you describe what these patterns look for?
Escapes

- We've now seen several characters that have special meaning in regular expressions:
  - `/` marks the end of the expression
  - `[ ]` used to describe sets of characters
  - `{ }` used to specify number of matches, e.g. for sets
  - `\` denotes the start of a special set, e.g. `\w`, `\d`
- `, + ?` special symbols
- What happens if you want to use one of these characters in a pattern?
  - e.g. what if you want to find `<\/>`?
- Answer: you need to "escape it" with a backslash:
  - \ in front of a special character tells Ruby to use the character itself

Example: Headlines

- We now have most of the tools we need to look for the headlines in this text.
- Our goal is to find strings surrounded by HTML tags `<B>` and `</B>`.
- There can be any characters (letters, digits, punctuation) between the tags.
- One way to write the pattern:

```
/<B>.*</B>/
```

```
Sports</B>
Low Payroll and High Hopes for Brewers</B>
Tough Park for Hitters? Mets Don't Think So</B>
Seven-R.B.I. G--- Go From Boston Stale</B>
Lilly and Blue
Dodgers' Reser
```

Note the `<B>` and `</B>` are part of the pattern, so they are part of the output.

Places to Use Regular Expressions

- In most of the previous examples we've seen regular expressions used as parameters to the `scan` method of the string class.
- Ruby lets you use a regular expression almost anywhere you use a string.
- Example: `index`
  - the `index` method returns the location of the start of a pattern or string.

```
>> s.index("<B>")
=> 311

>> s.index(/<B>.*<\/B>/)
=> 311

>> s.index(/\w{6}/)
=> 391
```

More Places to Use Regular Expressions

- A regular expression can be used to split a string into smaller pieces:

```
>> s[0..39]
=> "<html><head><title>Start Page</title><n"

>> lines = s.split("\n")
>> lines[0..2]
=> ["<html>", "<head>", "<title>Start Page""

>> s.split(/\b/)
=> ["", "html", ">\n<", "head", ">\n<", ...]
```

- What do you think this will do?

```
>> parts = s.split(/<B>.*<\/B>/)
```

- How many items are in parts?
Still More Places to Use Regular Expressions

- The `gsub` method does a "global substitution"
- It takes two arguments:
  - a pattern -- either a string or a regular expression
  - a replacement string
- Every time it finds a match for the pattern it replaces the match

```ruby
>> s.gsub!(/\d{6}/,"gray")
>> s.scan(/COLOR=".*?"/)
=> ["COLOR="ffffff", "COLOR="#006633"",
    "COLOR="ffffff", "COLOR="#ffffff"",
    "COLOR="gray\", ...]
```

Evidently not all colors in this document are preceded by #...

(Note: Ruby quotes the " characters appearing in the output strings...)

Implementation

- Searches based on regular expressions are very similar to the automaton based searches in the exact match problem
- the regular expression corresponds to the pattern to search for
- a preprocessing step creates an automaton from the expression
- the text is scanned one letter at a time
- each letter from the text leads to a state transition in the automaton
- when entering state 1 store the current text index so you remember where the pattern was found in the text

Limitations

- The fact that regular expressions correspond to finite automata implies there is a limit to the complexity of patterns they can find
  - this is also true of Ruby's extended regular expressions
- Example:
  - can you write a regular expression that finds the parts of an HTML table?
    - not if the table can contain other tables
  - To look for nested substructures like this a machine would need a memory -- e.g. a stack -- and the automaton would need to be a PDA

In-Class Lab

- time (e.g. 5:22AM) -- extracting hour, minute
- urls (http://www.xxx.yyy)
- quoted strings
e.g. COLOR="#666666"
- font names
e.g. <FONT FACE="Verdana,Arial,Helvetica,sans-serif"...>
- SPAM words
  - mort.gage
  - d.ipl*ma

In-Class Lab (Diagram)

- time (e.g. 5:22AM) -- extracting hour, minute
- urls (http://www.xxx.yyy)
- quoted strings
e.g. COLOR="#666666"
- font names
e.g. <FONT FACE="Verdana,Arial,Helvetica,sans-serif"...>
- SPAM words
  - mort.gage
  - d.ipl*ma

In-Class Lab (Diagram)

- time (e.g. 5:22AM) -- extracting hour, minute
- urls (http://www.xxx.yyy)
- quoted strings
e.g. COLOR="#666666"
- font names
e.g. <FONT FACE="Verdana,Arial,Helvetica,sans-serif"...>
- SPAM words
  - mort.gage
  - d.ipl*ma

In-Class Lab (Diagram)

- time (e.g. 5:22AM) -- extracting hour, minute
- urls (http://www.xxx.yyy)
- quoted strings
e.g. COLOR="#666666"
- font names
e.g. <FONT FACE="Verdana,Arial,Helvetica,sans-serif"...>
- SPAM words
  - mort.gage
  - d.ipl*ma
Advanced Topic: Memory

- Recall the expression that extracts headlines
  - headlines are strings between `<B>` and `</B>`
    ```ruby
    >> s.scan(/<B>.*</B>/)
    => ["<B>Sports</B>", "<B>Low Payroll and High Hopes for Brewers</B>", ...]
    ```
- There is a small problem here -- the `<B>` and `</B>` are included in the match
- How can we write an expression that tells Ruby the pattern we want is between the delimiters, but not to include the delimiters?
- Answer: use groups in the expression
  - a group is a part of a regular expression surrounded by parentheses
  - For the headline example, the expression to use is:
    ```ruby
    /<B>(.*)</B>/
    ```

Review

- A regular expression is a string used as a pattern
- A regular expression search looks for instances of the pattern in a text
- Patterns are built up from
  - letters (which stand for themselves)
  - character classes (\d, \w, \s), anchors (^, $, \b), sets ([...])
  - other special symbols (+, *, .., {, } )
- Ruby provides several opportunities to use regular expressions
  - `scan`, `index`, `split`, `gsub`, and other methods that have string arguments
- Skills:
  - give a short description of the kinds of strings a regular expression matches
  - write a regular expression given a description of a pattern to search for
  - labs, problem sets, and exams will not use memory or other advanced features

Memory (cont’d)

- When an expression contains one or more groups, `scan` returns only the parts that match the groups
- But there is a new complication:
  - an expression can have any number of groups
  - for an expression with *n* groups `scan` returns *n* strings for each match
  - the result is a list of lists
    ```ruby
    >> s.scan(/\d{1,2}:\d{2}[AP]M/)  
    >> s.scan(/(\d{1,2}):(\d{2})[AP]M/)  
    => [["5", "48"], ["5", "48"], ["5", "22"], ["5", "22"], ["5", "22"]]
    ```